Comput. Math. Applic. Vol. 13, No. 5/6, pp. 413–430, 1987
Printed in Great Britain. All rights reserved

0097-4943/87 $3.00 + 0.00
Copyright © 1987 Pergamon Journals Ltd

FUNCTIONAL TEST GENERATION USING BINARY DECISION DIAGRAMS†

M. S. ABADIR[1] and H. K. REGHBATI[2]

[1]Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089-0781, U.S.A.
[2]School of Computing Science, Simon Fraser University, Burnaby, B.C. V5A 1S6, Canada

Abstract—This paper proposes an extension to the D-algorithm, for integrated circuits described using binary decision diagrams. An LSI/VLSI circuit is modeled as a network of interconnected modules such as counters, registers, ROMs, RAMs, decoders, and multiplexers. The individual modules are described at the functional level using binary decision diagrams. A fault model is developed at the functional level quite independent of the implementation details of the individual modules in the chip. A generalization of the D-algorithm is proposed which takes the module-level model and the functional description of the modules as parameters, and generates tests to detect the faults in the fault model. Algorithms which perform the basic operations of the test generation procedure, on the functional modules, are also presented.

1. INTRODUCTION

The increasing use of LSI/VLSI circuits, and the highly reliable operation required in some applications, has created growing demands to develop new methods to generate efficient, thorough and cost-effective tests to detect faults in LSI/VLSI circuits. Realizing the fact that test generation has to be done economically, the major thrust of the past decade has been in the direction of automatic test generation.

Several classic approaches, based on algorithmic methods, have been developed for automatic test generation [6, 8, 10, 13, 15]. Most of these techniques require a detailed specification of precisely how the unit under test (UUT) has been implemented. Tests are then generated which will detect various postulated faults within this given implementation. Although this approach worked very well for SSI and MSI, they are extremely inefficient for testing LSI/VLSI circuits. Not only because the number of gates and interconnections become prohibitive, but in most cases, the implementation details of the UUT are not disclosed by the manufacturer. Thus, the problem of generating tests directly from the functional description of the UUT has become increasingly important.

Lately, two interesting approaches for microprocessor testing have been proposed [14, 18]. The instructions of the microprocessor are mapped into graph transitions modeling register transfers. The main limitation of these approaches is their restriction to microprocessors.

Attempts have also been made to extend the D-algorithm to circuits described using computer hardware description languages [11, 12, 17]. Essentially, a fault, from a specified fault model, is inserted into the description and a mechanism is used to propagate the fault forward. These approaches were motivated by the same reasons that motivated our work. However, we favored the use of binary decision diagrams, to describe the UUT, over the use of description languages, because the former models the circuit at a level that is closer to its actual implementation. For a comparison between these two description techniques and their effect on testing, the interested reader can refer to [3].

Another extension of the D-algorithm for functional primitives proposed in [9] has the disadvantage that it assumes a representation of any logic circuit using a number of built-in primitives.

In this paper, we are concerned with formulating a sound theoretical foundation for automatic test generation procedures for testing LSI/VLSI circuits. These procedures should treat the

†This research is supported by the Natural Sciences and Engineering Research Council of Canada under Grant No. A0743.

organization and the functional description of the chip as parameters. This is necessary because of the very large variety of LSI/VLSI chips available today in the market.

The test generation process usually encompasses three main activities:

(1) Selecting a good description model at a suitable level.
(2) Developing a fault model to define the types of faults that will be considered.
(3) Generating tests to detect all the faults in the fault model.

Clearly, test generation rests heavily on models, i.e. description models, and fault models. It is, therefore, very important to provide accurate modeling. In the next section we present a general model for LSI/VLSI circuits at the module level. We will also examine the functional description tool that will be used to describe the individual modules—*binary decision diagrams* [4, 5]—and point out the reasons behind selecting this particular description technique and how it simplifies various testing problems.

A functional level fault model capable of describing faulty behavior at a higher level is presented in Section 3. In Section 4, our test generation procedure is presented. The procedure takes the module level model of the LSI/VLSI chip and the functional description of its modules as parameters and generates tests to detect faults in the fault model. As in the D-algorithm, there are three basic operations employed in our test generation procedure: implication, D-propagation and line justification. Algorithms that perform these basic operations on functional modules are presented in Section 5.

2. A MODEL FOR LSI/VLSI CIRCUITS

For LSI/VLSI chips which have thousands of gates and flip-flops, using the gate and flip-flop level model for test generation purposes may not be feasible. On the other hand modeling an LSI/VLSI circuit as a black-box performing a specified input/output mapping is also infeasible, as most such circuits perform very complicated functions. Hence, the model needed should be at a level between these two extremes.

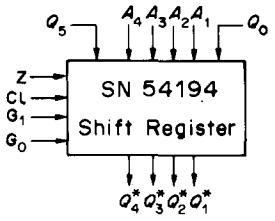
In view of the above requirements, it is appealing to approach the problem at a level close to that of the physical modules in the circuit. The module-level description model has been considered previously by a number of authors [7, 9, 16]. At this level, the system under consideration is modeled as a network of interconnected modules such as counters, registers, decoders, multiplexers etc. Each one of these modules has a well defined function. However, the selection of a good description tool to describe the functions performed by each module is the main issue affecting the success of the model.

The following is a set of requirements that must be satisfied by a module-level model suitable for generating tests for LSI/VLSI circuits:

(1) The model should be able to describe all the different types of modules that could be found in an LSI/VLSI circuit.
(2) The model should be able to support a good fault model.
(3) The model should be suited for automation.

In view of the above requirements we will now present our proposed model for LSI/VLSI circuits. An LSI/VLSI circuit, **C**, is modeled as a network of interconnected functional modules such as counters, registers, shifters, multiplexers, decoders etc. The interconnection structure of these modules is well defined in the model. **C** has a set of *primary input* lines which are directly controllable from the outside world and they can be set to any logical value. The circuit's set of primary inputs may have a clock signal† as one of its members. **C** has also a set of *primary output* lines which are directly accessible to the exterior of **C**.

†LSI/VLSI circuits that have more than one clock signal as primary inputs can still be modeled by our model. However, for simplicity reasons, we assume that the UUT has a unique clock signal. This does not exclude the possibility of having a counter module inside the circuit to generate other clock signals with cycles that are multiples of the fundamental clock cycle. It must be noted that, LSI/VLSI circuits built using MOS technology require two clock signals with different phases. However, the effect of these two phase clocks is equivalent to the effect of one clock. Hence, they could be modeled as one functional clock.



z	Cl	C_1	G_0	Functional behavior
1	x	x	x	Clear every bit of the register.
0	0	x	x	Hold the register contents.
0	1	1	1	Hold the register contents.
0	1	0	0	Load the values of (A_1, A_2, A_3, A_4) in parallel.
0	1	0	1	Shift the register content one bit to the left. Q_0 is fed into the register from the right.
0	1	1	0	Shift the register content one bit to the right. Q_3 is fed into the register from the left.

Fig. 1. A 4-bit bidirectional shift register and its functional behavior.

The modules in **C** can be classified as combinational or sequential, according to the function performed by each module. Each memory element in the circuit is described by two variables; the current state variable, and the next state variable. The latter variable will always be marked with an asterisk (*). This way of description will allow us to treat sequential modules as combinational ones, while defining their functional operations. This issue will be clarified in the next section. We assume that each sequential module, **S**, in the ciruit is synchronous i.e. all sequential modules have the clock signal as one of their inputs.

We assume that the state of each internal memory element in **S** is one of its outputs. This assumption can be easily justified for most sequential modules usually encountered in LSI/VLSI circuits, such as latch registers, shift registers, and counters. Other sequential modules that don't satisfy this condition can be modelled as a combination of smaller functional modules, each of which satisfies our assumption. For example, a RAM or a ROM can be modeled as a large addressable register connected to a multiplexer [1]. This strategy is being used in accordance with our goal of breaking a complicated chip into small modules each of which can be tested easily. Clearly, sequential modules with (internally) observable memory elements are much simpler to test than the ones with hidden memory elements.

In our model, the circuit can have one or more feedback loops in the interconnections between the modules. However, we assume that there is at least one memory element in any feedback loop in the network. This assumption is necessary to avoid the creation of new memory elements in the circuit, and it is easily justified for all synchronous sequential LSI/VLSI circuits.

Each module in **C** is modeled as a black box, realizing a number of functions defined using a set of *binary decision diagrams* [4]—a functional description tool introduced by Akers in 1978. A binary decision diagram is nothing more than a concise means for completely defining the logical operation of one or more digital functions in an implementation-free form. The information usually found in an IC catalog is sufficient to derive the set of binary decision diagrams describing the functions performed by the different modules in that device.† These diagrams—like truth tables and state tables—are amenable to extensive logical analysis. However, they don't have the unpleasant property of growing exponentially with the number of variables involved as in the case of truth tables and state tables. Moreover, the diagrams can be stored and processed very easily in a digital computer. For more details on binary decision diagrams, and their use in the area of functional testing, the reader should consult the original papers by Akers [4, 5]. The following two examples illustrate how binary decision diagrams are used to describe functional modules.

Example 1. Consider the SN54194 4-bit bidirectional shift register shown in Fig. 1. In normal operation this register can perform five operations: clear, hold, parallel load, shift right and shift left. The functional behavior of this device is determined by the values of four control signals (Z, Cl, G_1, G_0) as specified by the table accompanying Fig. 1.

The shift register has 10 external inputs and 4 output lines. The next state of the ith bit of the register is denoted by Q_i^* while the current state value of the same bit is denoted by Q_i. Figure 2 shows the binary decision diagram which completely defines the operation of the ith bit of the register. Each node in the diagram is associated with a binary variable and there are two branches

†There are procedures to generate the binary decision diagram of a function from its truth table or from its Boolean expression. Similar procedures can easily be constructed for functions described by state tables, or by a hardware description language.

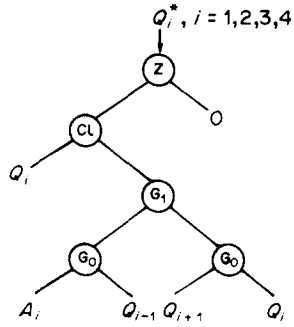


Fig. 2. Binary decision diagram for the shift register.

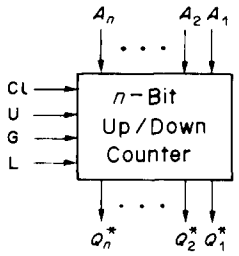
coming out from each node, the right branch is the one branch, while the left branch is the zero branch. According to the value of the node variable one of the two branches will be selected when processing the diagram. By entering the diagram at the node indicated by the arrow labelled with Q_i^*, and then proceeding through the diagram following the appropriate branches until a terminal value is reached, the value of Q_i^* is determined.

Example 2. Consider the n-bit up/down counter shown in Fig. 3. The functional behavior of this device is determined by the values of four control signals (Cl, U, G, L) as specified by the table accompanying Fig. 3.

The next state of bit i of the counter is denoted by Q_i^* while the current state value of the same bit is denoted by Q_i. Figure 4 shows the binary decision diagram which completely defines the operation of this counter. Note that there are $2n-4$ auxiliary variables c_i and b_i for $i = 2,3 \ldots, n-1$ used to simplify the description. All node variables other than input variables define auxiliary functions, where $c_i(b_i)$ represents the carry (borrow) signal resulting from the ith state of the counter during the count up (count down) mode of operation.

Functional testing with binary decision diagrams

Functional testing can be defined as the process of verifying that a given module does what it is supposed to do [5]. For example, if the module is a cell in the memory, the functional testing



Cl	U	G	L	Mode of operation
x	x	x	0	Parallel load
x	x	1	1	Hold
0	x	0	1	Hold
1	1	0	1	Count up
1	0	0	1	Count down

Fig. 3. An n-bit up/down counter and its functional behavior.

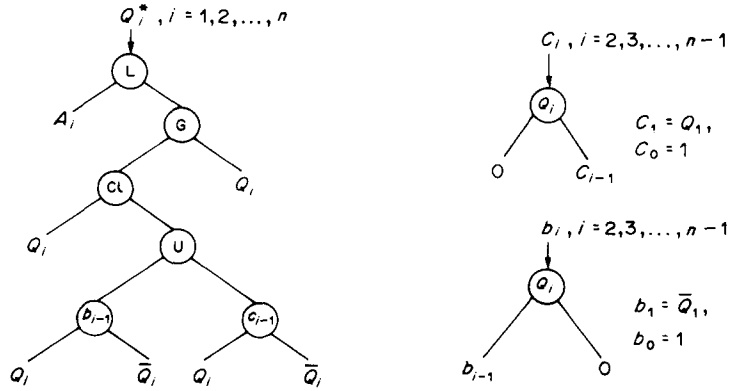


Fig. 4. Binary decision diagram for the counter.

process must check that the read operation reads the cell contents, the write operation writes the desired value in the cell, the cell can hold its information, etc. Thus, we can reformulate the definition of functional testing as the process of checking the performance of a module in its various *modes of operation.*

Clearly, functional testing requires a description of the module's performance in each mode of operation. We will refer to the specification of one mode of a module's performance as an *experiment* [5]. In the next example we will illustrate how the module's complete set of experiments can be generated from its binary decision diagram.

Example 3. Consider the diagram of Fig. 2. Every possible path starting at Q_i^* represents one mode of operation, or equivalently specifies an experiment. It follows that, if we trace all the possible paths from Q_i^* to an exit value or to an exit variable, we will obtain automatically a complete and disjoint set of experiments for Q_i^*. Each experiment is formulated by simply recording the branch values of the variables involved in the path. The experiment's output is obtained by recording the exit value or the value of the exit variable. All the variables which are not involved in a certain path are assigned xs (an x means a *don't care* value). Hence, a complete set of experiments describing the ith bit of the shift register is shown in Table 1.

As mentioned before, the problem of testing a given module can be described as the problem of validating the module's performance in its different modes of operation, i.e. we partition the problem into small subproblems, each of which involves validating one mode of operation as described by the associated experiment. Of course, the relatively simple nature of an individual experiment makes the problem much easier to handle.

3. THE FAULT MODEL

Two classes of faults are considered in our model. *Class 1 faults* consist of all single stuck-at faults that affect the module's input lines, output lines, or internal memory elements. A module's functional fault which adversely affects the execution of one of its experiments is said to be a *class 2 fault.*

For example, consider a module **M** having f as one of its output functions, and assume that the following experiment describes f in one of its modes of operation:

$$f(1, 0, 0, 0, x, x) = 0$$

The functional fault **F** that changes the output of the above experiment to 1 instead of 0 is a class 2 fault. Thus, for a given module the number of class 2 faults equals the number of experiments describing the module outputs and auxiliary variables. By definition, it is clear that a complete test for this class of faults can be constructed from the complete set of experiments describing the module. It should be noted that there is no need to consider faults that adversely affect two or more experiments of the same module function, because these faults will be detected while testing for class 2 faults. This is because the experiments are disjoint.

The fault model. At any given time we allow the presence of only one fault, either from class 1 or from class 2, in the UUT. If we allowed multiple faults in different modules, the problem would become extremely complex. However, most multiple faults (i.e. those which do not mask each

Table 1. The complete set of experiments for the ith bit of the shift register

Z	Cl	G_1	G_0	Q_{i-1}	Q_i	Q_{i+1}	A_i	Q_i^*	
1	x	x	x	x	x	x	x	0	Clear
0	0	x	x	x	0	x	x	0	Hold
0	0	x	x	x	1	x	x	1	
0	1	1	1	x	0	x	x	0	
0	1	1	1	x	1	x	x	1	
0	1	0	0	x	x	x	0	0	Load
0	1	0	0	x	x	x	1	1	
0	1	0	1	0	x	x	x	0	L Shift
0	1	0	1	1	x	x	x	1	
0	1	1	0	x	x	0	x	0	R Shift
0	1	1	0	x	x	1	x	1	

other) will be detected while testing for single faults. The single fault assumption is included in most practical fault models. It is justified if the module's failures are independent, and if the circuit is tested relatively often. It must be noted, however, that any single functional fault belonging to class 2 faults may correspond to one or more physical or logical faults. Thus, our fault model covers a wide range of physical and logical faults [3].

It should be emphasized that other classes of faults can be easily incorporated in our fault model as the need arises. Once the effect of a fault on the behavior of the circuit has been defined, the same techniques that will be discussed in the following sections can be used to generate a test for that fault.

4. TEST GENERATION PROCEDURE

Let **M** be a module in the circuit under test, **C**. Assume that we want to generate a test to detect a possibly existing fault in module **M**. Following the path sensitization testing technique [8, 15], the first step is to excite the fault, i.e. to specify the values of some of the module input variables so as to generate an error signal at one of the module output lines (*fault excitation*). The next step would be to propagate that error signal through other modules to at least one of the observable outputs of **C** (*D-propagation*). Finally, the last step involves justifying the signal values specified in the previous two steps (*line justification*).

Next we will describe how the various faults covered by our fault model can be excited.

Fault excitation process

Let **F1** be any stuck-at-j class 1 fault, where $j = 0$ or 1, affecting module **M**. By definition, two cases have to be considered:

(1) **F1** is affecting an output line of M. In this case any experiment which sets that output line to logic value $\bar{j}$ can be used as a primitive D-cube [8, 15] for **F1**. (Note that stuck-at faults affecting the internal memory elements of **M** are covered under this category, since the state of each memory element is assumed to be a module output.)

(2) **F1** is affecting an input line of **M** which is one of the primary inputs of **C**. In this case if we apply $\bar{j}$ to that input line, an error signal D (0 instead of 1) or $\bar{\mathrm{D}}$ (1 instead of 0) will exist in that input line. Hence the fault is excited at its site. Note that if line 1 is an input to a module but it is not one of the primary inputs of **C**, then line 1 must be an output from another module **K** in **C**. Thus stuck-at faults affecting line 1 can be excited as described in (1) above.

On the other hand, let **F2** be a class 2 fault affecting experiment e_i of module **M**. By definition, e_i with an appropriate error signal on its output is the primitive D-cube of **F2**.

Next we will define some of the terms that will arise frequently in our discussion:

Circuit variables. Any module input variable, output variable, auxiliary variable, or internal memory element variable is a circuit variable. Note that, memory elements are described by two variables: the current state variable, and the next state variable. The circuit variables are numbered $1, 2, \ldots, v$, where v is the number of variables in the circuit.

The *test cube* of the circuit **C** is a vector of size v whose ith element specifies the logical value 0, 1, D, $\bar{\mathrm{D}}$, or x associated with circuit variable number i at a given instant of time. Each single step in the procedure to be presented will have an effect on the test cube in one way or another. At the end of the procedure, the values assigned to the primary input variables in the test cube represent the test vector generated. At the start of the procedure, the test cube will be initialized with the circuit's initial state†.

D-frontier. The set of all modules which have Ds or $\bar{\mathrm{D}}$s on their input variables or on their current state variables, an x on their output, and for which the D-propagation operation has not yet been performed.

---

†The problem of circuit initialization is a very complex problem which, as yet, has not been adequately solved. One way to solve this problem is to start the test generation process by testing the circuit in some of its modes of operation that do not depend on the initial state of the circuit. For instance, the clear mode of operation of a register and the write mode of operation of a memory cell [1].

The term *time frame* will be used to denote the period of time between two clock cycles. Hence, during a single time frame, any memory element can change its state only once. We assume that the primary inputs of **C** are assigned values at the start of each time frame and they will remain unchanged during any time frame. It must be noted that some faults may require multiple vector tests. The test generation to be presented next is designed to minimize the number of test vectors generated to detect a given fault.

Procedure: TG

(1) Set t to 1. (The variable t represents the time frame number.)

(2) Excite the fault under consideration. This produces an error signal D or $\bar{D}$ somewhere in the circuit. If a choice exists, select an arbitrary one initially.

(3) Perform *implication* for the test cube obtained in step (2). The implication procedure determines the circuit variables which must be changed unambiguously from x to 0, 1, D or $\bar{D}$ due to the fact that some other circuit variables have just been specified to be 0, 1, D or $\bar{D}$. An inconsistency may be encountered during implication if a value is implied on a variable y which has previously been specified to a different value. If this happens, we must backtrack to the last point a choice existed, resetting all variables to their values at that point and starting again with the next choice.

(4) *D-propagation.* Select a module from the D-frontier and try to propagate an error signal to one of its outputs. If this is not possible, select another module from the D-frontier and repeat this step. If successful, update the test cube and the D-frontier appropriately.

(5) Perform implication for the test cube derived in (4).

(6) Repeat steps (4) and (5) until one of the following occurs: (a) The faulty signal has been propagated to one of the circuit's primary outputs—go to step (8)—or (b) the D-frontier is found to be empty. In this case, we must backtrack. However, if at least one next state variable in the current test cube carries an error signal, then save the test cube in a list H_t before backtracking.

(7) This step will be executed if all the choices have failed to propagate the error signal to a primary output at time frame t. Check the H_t list, if it is empty and $t > 1$, then we must backtrack to the previous time frame. Decrement t by 1 and execute this step again. If the H_1 list is found to be empty then no test can be found for the fault considered and the procedure terminates. If the H_t list is not empty, then we will try to propagate the error signal to one of the primary outputs of **C** in time frame $t + 1$. Select a cube from the H_t list, this cube describes the circuit at time frame t and it will be denoted by TC_t. From TC_t derive the initial test cube at time frame $t + 1$ and increment t by 1. If the new initial test cube is similar to the initial test cube of any previous time frame, then backtrack, otherwise go to step (3).

(8) *Line justification.* Execution of (1)–(7) may result in specifying an output value of a module **M** but leaving some of its inputs unspecified. The inputs to such a module are now specified to justify the output value. Implication is then performed on the new test cube. This step is repeated until all the modules have been justified (in all time frames). Backtracking may again be required. □

The procedure described above will generate a test for any fault covered by our fault model if such a test exists. This test will either take the form of a single vector or a multiple vector test sequence. Note that, the upper bound of the number of test vectors required for any fault is 4^n, where n is the number of state variables in the UUT. This is because the initial test cube at any time frame is restricted to be unique and there are only 4^n such unique states.

The structure of our test generation procedure is similar to the structure of the D-algorithm, but at a different scale. As in the D-algorithm, our procedure employs three basic operations namely implication, D-propagation, and line justification. These operations have to be performed on functional modules. The algorithms which perform the three basic operations on the functional modules are presented in the next section.

5. THE BASIC OPERATIONS

Terminology and definitions

Let **M** be a module having input **X** and output **Y**, where **X** and **Y** are vectors of size n_x and n_y, respectively. The module is described at the functional level using a set of experiments **E**. Any

module output $y \in \mathbf{Y}$ is described by a set of experiments $\mathbf{E}_y$ which is a subset of $\mathbf{E}$. We recall that it is possible to use some auxiliary functions to simplify the description of a module (see Example 2). Assume that $\mathbf{M}$ has a set of auxiliary functions $\mathbf{Z}$, where $\mathbf{Z}$ is a vector of size n_z. Each *auxiliary variable* $z \in \mathbf{Z}$ is described by a set of experiments $\mathbf{E}_z$ which is a subset of $\mathbf{E}$. Let $\mathbf{W}$ be a vector that contains all the output variables and all the auxiliary variables of the module $\mathbf{M}$. Hence, the size of $\mathbf{W}$, denoted as n_w, equals $n_y + n_z$. We will use the term *module variable* to denote either a module's input, output, or auxiliary variable. The module variables are numbered as follows:

(1) Number the input variables $\mathbf{X}$ with the integers $1, 2, \ldots, n_x$.

(2) Number the output and auxiliary variables $\mathbf{W}$ with the integers $n_x + 1, n_x + 2, \ldots, (n_x + n_w)$ in such a way that the number associated with an output or an auxiliary variable is larger than the number associated with any of its input variables. Clearly, this is possible since we are dealing with synchronous circuits.

Definition. A *module test cube*, $\mathbf{T}$, is a vector of length $n_x + n_w$ whose ith element specifies the value (0, 1, D, $\bar{\mathrm{D}}$ or x) associated with module variable number i at a given time.

The experiments in $\mathbf{E}$ are organized into n_w subsets. Thus, $\mathbf{E} = \{e_1, e_2, \ldots, e_{n_w}\}$ where e_i is the set of experiments that describes the variable w_i. Any experiment in e_i, for $i = 1, 2, \ldots, n_w$, is a vector of size $n_x + n_w$. Note that all the module variables that do not affect the output value of an experiment are assigned xs.

To illustrate the above definitions consider the following example.

Example 4. Consider a 4-bit up/down counter of the type described in Example 2. The counter has 8 external inputs, and 4 internal memory elements, hence $n_x = 12$. This counter also has 4 output variables $\{Q_1^*, Q_2^*, Q_3^*, Q_4^*\}$ and 4 auxiliary variables $\{c_2, b_2, c_3, b_3\}$. The module's 20 variables could be numbered and arranged accordingly in the module test cube as follows:

$$\mathbf{T} = (\mathrm{Cl}, \mathrm{U}, \mathrm{G}, \mathrm{L}, A_1, A_2, A_3, A_4, Q_1, Q_2, Q_3, Q_4, Q_1^*, Q_2^*, c_2, b_2, Q_3^*, c_3, b_3, Q_4^*)$$

In order to generate a complete set of experiments for this counter, we will trace out the various paths which start at each of the 4 outputs (Q_i^*, $i = 1, 2, 3, 4$) as well as the various paths which start at each of the auxiliary variables (c_i and b_i *for* $i = 2, 3$). The complete set of experiments $\mathbf{E}$ describing the counter module is shown in Table 2. Each individual experiment is of size 20. The set $\mathbf{E}$ is partitioned into 8 subsets $\{e_1, e_2, \ldots, e_8\}$. Any subset e_i describes the ith variable in $\mathbf{W}$, or equivalently the $(i + 12)$th variable in $\mathbf{T}$. The total number of experiments in $\mathbf{E}$ is 64.

Performing one of the three basic operations on a module $\mathbf{M}$ will result in generating a new cube for that module. If more than one solution exists, all the possible solutions must be generated. The reason behind this strategy is to avoid performing the same operation on a module test cube more than once. Thus, if one solution leads to an inconsistency somewhere in the circuit the next solution—if one exists—is selected and the test generation process continues as described in the previous section. Next we will consider each basic operation separately.

Implication

The implication problem can be defined as follows: *For a module* $\mathbf{M}$ *determine which of its variables currently at value* x *must be changed unambiguously to* 0, 1, D *or* $\bar{\mathrm{D}}$ *due to the fact that some other variables of* $\mathbf{M}$ *have been specified to* 0, 1, D, *or* $\bar{\mathrm{D}}$.

Consider one of the module outputs or auxiliary variables w_i and assume that it is not specified in the module test cube. We want to determine whether some of the specified module variables uniquely imply the value of w_i or not. Let the set e_i be the set of experiments which describes w_i. To solve this subproblem, the module test cube $\mathbf{T}$ is intersected with each experiment in the set e_i. If all the experiments in the set e_i that can be intersected with $\mathbf{T}$ assign the same value (1 or 0) to w_i, then this value is implied on w_i. Otherwise nothing is implied. This process is called *forward implication* and it is performed on all the unspecified outputs or auxiliary variables in $\mathbf{T}$.

After performing forward implication, the implication process is performed backward. Consider an output or auxiliary variable w_i which is specified in the module test cube. Let the set e_i be the set of experiments that describes w_i. We want to determine if the value of w_i together with other specified values in the module test cube uniquely imply the value of any of the unspecified variables.

Table 2. The complete set of experiments for the up/down counter

	X												W							
	Cl	U	G	L	A_1	A_2	A_3	A_4	Q_1	Q_2	Q_3	Q_4	Q_1^*	Q_2^*	c_2	b_2	Q_3^*	c_3	b_3	Q_4^*
e_1	x	x	x	0	0	x	x	x	x	x	x	x	0	x	x	x	x	x	x	x
	x	x	x	0	1	x	x	x	x	x	x	x	1	x	x	x	x	x	x	x
	x	x	1	1	x	x	x	x	0	x	x	x	0	x	x	x	x	x	x	x
	x	x	1	1	x	x	x	x	1	x	x	x	1	x	x	x	x	x	x	x
	0	x	0	1	x	x	x	x	0	x	x	x	0	x	x	x	x	x	x	x
	0	x	0	1	x	x	x	x	1	x	x	x	1	x	x	x	x	x	x	x
	1	1	0	1	x	x	x	x	0	x	x	x	1	x	x	x	x	x	x	x
	1	1	0	1	x	x	x	x	1	x	x	x	0	x	x	x	x	x	x	x
	1	0	0	1	x	x	x	x	0	x	x	x	1	x	x	x	x	x	x	x
	1	0	0	1	x	x	x	x	1	x	x	x	0	x	x	x	x	x	x	x
e_2	x	x	x	0	x	0	x	x	x	x	x	x	x	0	x	x	x	x	x	x
	x	x	x	0	x	1	x	x	x	x	x	x	x	1	x	x	x	x	x	x
	x	x	1	1	x	x	x	x	x	0	x	x	x	0	x	x	x	x	x	x
	x	x	1	1	x	x	x	x	x	1	x	x	x	1	x	x	x	x	x	x
	0	x	0	1	x	x	x	x	x	0	x	x	x	0	x	x	x	x	x	x
	0	x	0	1	x	x	x	x	x	1	x	x	x	1	x	x	x	x	x	x
	1	1	0	1	x	x	x	x	0	0	x	x	x	0	x	x	x	x	x	x
	1	1	0	1	x	x	x	x	0	1	x	x	x	1	x	x	x	x	x	x
	1	1	0	1	x	x	x	x	1	0	x	x	x	1	x	x	x	x	x	x
	1	1	0	1	x	x	x	x	1	1	x	x	x	0	x	x	x	x	x	x
	1	0	0	1	x	x	x	x	0	0	x	x	x	1	x	x	x	x	x	x
	1	0	0	1	x	x	x	x	0	1	x	x	x	0	x	x	x	x	x	x
	1	0	0	1	x	x	x	x	1	0	x	x	x	0	x	x	x	x	x	x
	1	0	0	1	x	x	x	x	1	1	x	x	x	1	x	x	x	x	x	x
e_3	x	x	x	x	x	x	x	x	x	0	x	x	x	x	0	x	x	x	x	x
	x	x	x	x	x	x	x	x	0	1	x	x	x	x	0	x	x	x	x	x
	x	x	x	x	x	x	x	x	1	1	x	x	x	x	1	x	x	x	x	x
e_4	x	x	x	x	x	x	x	x	0	0	x	x	x	x	x	1	x	x	x	x
	x	x	x	x	x	x	x	x	1	0	x	x	x	x	x	0	x	x	x	x
	x	x	x	x	x	x	x	x	x	1	x	x	x	x	x	0	x	x	x	x
e_5	x	x	x	0	x	x	0	x	x	x	x	x	x	x	x	x	0	x	x	x
	x	x	x	0	x	x	1	x	x	x	x	x	x	x	x	x	1	x	x	x
	x	x	1	1	x	x	x	x	x	x	0	x	x	x	x	x	0	x	x	x
	x	x	1	1	x	x	x	x	x	x	1	x	x	x	x	x	1	x	x	x
	0	x	0	1	x	x	x	x	x	x	0	x	x	x	x	x	0	x	x	x
	0	x	0	1	x	x	x	x	x	x	1	x	x	x	x	x	1	x	x	x
	1	1	0	1	x	x	x	x	x	x	0	x	x	x	0	x	0	x	x	x
	1	1	0	1	x	x	x	x	x	x	0	x	x	x	1	x	1	x	x	x
	1	1	0	1	x	x	x	x	x	x	1	x	x	x	0	x	1	x	x	x
	1	1	0	1	x	x	x	x	x	x	1	x	x	x	1	x	0	x	x	x
	1	0	0	1	x	x	x	x	x	x	0	x	x	x	x	0	0	x	x	x
	1	0	0	1	x	x	x	x	x	x	0	x	x	x	x	1	1	x	x	x
	1	0	0	1	x	x	x	x	x	x	1	x	x	x	x	0	1	x	x	x
	1	0	0	1	x	x	x	x	x	x	1	x	x	x	x	1	0	x	x	x
e_6	x	x	x	x	x	x	x	x	x	x	0	x	x	x	x	x	x	0	x	x
	x	x	x	x	x	x	x	x	x	x	1	x	x	x	0	x	x	0	x	x
	x	x	x	x	x	x	x	x	x	x	1	x	x	x	1	x	x	1	x	x
e_7	x	x	x	x	x	x	x	x	x	x	0	x	x	x	x	0	x	x	0	x
	x	x	x	x	x	x	x	x	x	x	0	x	x	x	x	1	x	x	1	x
	x	x	x	x	x	x	x	x	x	x	1	x	x	x	x	x	x	x	0	x
e_8	x	x	x	0	x	x	x	0	x	x	x	x	x	x	x	x	x	x	x	0
	x	x	x	0	x	x	x	1	x	x	x	x	x	x	x	x	x	x	x	1
	x	x	1	1	x	x	x	x	x	x	x	0	x	x	x	x	x	x	x	0
	x	x	1	1	x	x	x	x	x	x	x	1	x	x	x	x	x	x	x	1
	0	x	0	1	x	x	x	x	x	x	x	0	x	x	x	x	x	x	x	0
	0	x	0	1	x	x	x	x	x	x	x	1	x	x	x	x	x	x	x	1
	1	1	0	1	x	x	x	x	x	x	x	0	x	x	x	x	x	0	x	0
	1	1	0	1	x	x	x	x	x	x	x	0	x	x	x	x	x	1	x	1
	1	1	0	1	x	x	x	x	x	x	x	1	x	x	x	x	x	0	x	1
	1	1	0	1	x	x	x	x	x	x	x	1	x	x	x	x	x	1	x	0
	1	0	0	1	x	x	x	x	x	x	x	0	x	x	x	x	x	0	0	0
	1	0	0	1	x	x	x	x	x	x	x	0	x	x	x	x	x	x	1	1
	1	0	0	1	x	x	x	x	x	x	x	1	x	x	x	x	x	x	0	1
	1	0	0	1	x	x	x	x	x	x	x	1	x	x	x	x	x	x	1	0

As in forward implication, the module test cube **T** is intersected with each experiment in the set e_i. If the intersection exists, then the experiment will be placed on a list L. At the end, if all the experiments in **L** assign the same value to one of the unspecified input variables, then this value is implied on that variable. Otherwise nothing is implied. If the list **L** is found to be empty during

any backward implication process, then an inconsistency has been found and the implication process terminates as it is necessary to backtrack. The backward implication process is performed on all the specified output and auxiliary variables in **T**.

If the backward implication process implies one or more values on the module variables then forward implication must be performed once more. The two processes are iterated until no more values could be implied on the module test cube.

In the previous discussion we have only considered values implied by inputs of 0 or 1. However, it is also possible to have signal values implied by inputs of D or $\bar{D}$. This can be done in two steps, using a technique called *decomposition*. First perform implication on the fault-free case (i.e. substitute the Ds with ones and the $\bar{D}$s with zeros) and determine the values implied on the unspecified variables of the module. Next perform implication on the faulty case (i.e. the Ds are replaced with zeros and the $\bar{D}$s with ones) and determine the implied values on this case. The composite of the two sets of values obtained above is implied on the module variables. The composition operation is defined in Table 3.

In view of the above discussion we will formulate a general algorithm that will perform the implication operation. Since the solution to this problem is somewhat lengthy, it is convenient to separate some of the tasks involved in the solution into subalgorithms. The main algorithm, IMPLICATION, invokes a subalgorithm, ONE-CASE-IMPLY. This subalgorithm determines signal values implied by variables having the value 0 or 1. ONCE-CASE-IMPLY uses subalgorithms FORWARD-IMPLICATION and BACKWARD-IMPLICATION to determine signal values implied by the inputs on the outputs, and the outputs on the inputs, respectively. Now we will formulate the FORWARD-IMPLICATION subalgorithm.

Procedure FORWARD-IMPLICATION (T, E). Given a module test cube **T** and the module set of experiments **E**, this procedure performs forward implication.

```
Repeat for i = n_x + 1, n_x + 2, ..., n_x + n_w;
  If T(i) = x
  THEN DO;
    Flag ← 0; Implied-Value ← x;
    Repeat for every experiment α ∈ e_{i-n_x} while (Flag < 2);
      IF T ∩ α ≠ φ AND α(i) ≠ Implied-Value
      THEN DO;
        Implied-Value ← α(i);
        Flag ← Flag + 1;
        END;
      END;
    END;
  IF Flag = 1
  THEN T(i + n_x) ← Implied-value;
  END;
RETURN;
END;
```

There are two main local variables used in the above procedure: Flag, and Implied-Value. The latter is used to find the value that can be implied on any of the unspecified outputs or auxiliary variables. Flag is used to count the number of times the variable Implied-Value changes its value while considering one of the unspecified outputs or auxiliary variables. Flag = 1 indicates that the variable Implied-Value has changed its value only once from x (the initial value) to 0 or 1 and remained unchanged after that. Hence, this later value has to be implied on the variable under consideration. On the other hand, Flag = 2 indicates that it is not possible to imply the value of the variable under consideration.

Table 3. The composition operation

	Fault-free		
Faulty	0	1	x
0	0	D	x
1	$\bar{D}$	1	x
x	x	x	x

Next we will formulate the BACKWARD-IMPLICATION subalgorithm.

Procedure BACKWARD-IMPLICATION (T, E, Error-Flag). Given **T** and **E** as previously defined, this procedure performs backward implication. If an inconsistency has been found, the output variable Error-Flag will be set to 1. Otherwise, it will be 0.

```
Error-Flag ← 0;
Repeat for i = n_x + n_w, n_x + n_w − 1, ..., n_x + 1;
  IF T(i) ≠ x
  THEN DO;
    j ← 0;
    Repeat for every experiment α ∈ e_{i−n_x};
      IF T ∩ α ≠ φ
      THEN DO;
        j ← j + 1; L(j) ← α;
        END;
      END;
    IF j = 0
    THEN DO;
      Error-Flag ← 1; RETURN;
      END;
    Repeat for k = 1, 2, ..., i − 1
      IF T(k) = x
      THEN DO;
        Implied-Value ← x;
        Repeat for m = 2, 3, ..., j while (Implied-Value ≠ φ);
          Implied-Value ← Implied-Value ∩ L(m, k);
          END;
        IF Implied-Value ≠ φ
        THEN T(k) ← Implied-Value;
        END;
      END;
    END;
  END;
RETURN;
END;
```

Now we will formulate the ONE-CASE-IMPLY subalgorithm which uses the previous two subalgorithms.

Procedure ONE-CASE-IMPLY (T, E, Error-Flag). Given **T** and **E** as previously defined, this procedure determines all the signal values (0 or 1) that should be implied on the module unspecified variables due to the fact that other variables are specified to be 0 or 1. This procedure traces such signal determination both forwards and backwards through the module. If an inconsistency is found, the procedure terminates with the variable Error-Flag being set to 1.

```
Call FORWARD-IMPLICATION (T, E);
T* ← T;
Call BACKWARD-IMPLICATION (T, E, Error-Flag);
Repeat while (T* ≠ T AND Error-Flag ≠ 1);
  T* ← T;
  Call FORWARD-IMPLICATION (T, E);
  IF T* ≠ T
  THEN DO;
    T* ← T;
    Call BACKWARD-IMPLICATION (T, E, Error-Flag);
    END;
  END;
RETURN:
END;
```

The local variable T* is used to find out whether a forward or a backward implication process has implied new values on the module test cube or not. The process terminates if the execution of one of the two subalgorithms does not imply new signal values. (Each subalgorithm has to be executed at least once.)

Finally, we will present the main IMPLICATION algorithm which employs the previous three subalgorithms.

Procedure IMPLICATION (T, E, Error-Flag).

```
D-Flag ← 0;
IF T contains any error signal (i.e., a D or a D̄)
THEN DO;
  D-Flag ← 1;
  T1 ← The fault-free value of T (replace the Ds with ones and the D̄s with zeros);
  T2 ← The faulty value of T (replace the D's with zeros and the D̄s with ones);
  END;
IF D-Flag = 0
THEN Call ONE-CASE-IMPLY (T, E, Error-Flag);
ELSE DO;
  Call ONE-CASE-IMPLY (T1, E, Error-Flag);
  IF Error-Flag = 0
  THEN DO;
    Call ONE-CASE-IMPLY (T2, E, Error-Flag);
    IF Error-Flag = 0
    THEN T ← the composite of T1 and T2;
    END;
  END;
RETURN;
END;
```

The IMPLICATION procedure uses three local variables D-Flag, T1, and T2. D-Flag is used to indicate whether there is a D or a $\bar{D}$ in the module test cube or not. If D-Flag is set to 1, this means that a D or a $\bar{D}$ has been found. In this case two test cubes **T1** and **T2** are formulated using the original module test cube **T**. **T1** represents the module test cube in the fault-free case, while **T2** represents the module test cube in the faulty case. The implication process is performed on both **T1** and **T2** and the resulting cubes are composed together to obtain the final solution to the problem.

The next example will illustrate the operation of the implication algorithm.

Example 5. Consider the 4-bit up/down counter described in Examples 2 and 4. Assume that we want to perform implication on the counter module whose variables are partially specified as follows:

$$(\mathrm{Cl}, \mathrm{U}, \mathrm{G}, \mathrm{L}) = (1\ x\ x\ x), \qquad (A_1, A_2, A_3, A_4) = (x\ x\ 0\ x),$$
$$(Q_1, Q_2, Q_3, Q_4) = (x\ 1\ 0\ 0), \qquad (Q_1^*, Q_2^*, Q_3^*, Q_4^*) = (x\ x\ 1\ x).$$

Hence, the module test cube **T** is defined as follows:

$$\mathbf{T} = (1\ x\ x\ x\ x\ x\ 0\ x\ x\ 1\ 0\ 0\ x\ x\ x\ x\ 1\ x\ x\ x)$$

Note that there is no D or $\bar{D}$ signals in **T**, hence one case of implication will be considered using **T**. The following set of activities will occur:

(1) FORWARD-IMPLICATION: Intersecting **T** with e_1, e_2 and e_3 do not imply new values. However, intersecting **T** with e_4, e_6 and e_7 implies 0 on b_2, c_3, and b_3 respectively. FORWARD-IMPLICATION terminates with the following new test cube:

$$\mathbf{T} = (1\ x\ x\ x\ x\ x\ 0\ x\ x\ 1\ 0\ 0\ x\ x\ x\ 0\ 1\ 0\ 0\ x)$$

(2) BACKWARD-IMPLICATION: Nothing can be implied by intersecting **T** with both e_7 and e_6. However, intersecting **T** with e_5 results in implying the values of U, G, L, and c_2 to be 1, 0, 1 and 1, respectively. Intersecting **T** with e_4 does not imply new values, while intersecting **T** with e_3 results in specifying Q_1 to be 1. BACKWARD-IMPLICATION terminates with the following test cube:

$$\mathbf{T} = (1\ 1\ 0\ 1\ x\ x\ 0\ x\ 1\ 1\ 0\ 0\ x\ x\ 1\ 0\ 1\ 0\ 0\ x)$$

(3) FORWARD-IMPLICATION: Intersecting **T** with e_1 implies 0 on Q_1^*. Also, intersecting **T** with e_2 implies 0 on Q_2^*. Finally intersecting **T** with e_8 implies 0 on Q_4^* (the last unspecified output or auxiliary variable). The process terminates with the following module test cube:

$$\mathbf{T} = (1\ 1\ 0\ 1\ x\ x\ 0\ x\ 1\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 0)$$

(4) BACKWARD-IMPLICATION: No more values can be implied, hence the whole process terminates with the previous value of **T**.

D-propagation

The D-propagation problem can be defined as follows: *Given a* D *or* $\bar{D}$ *value at one of the* ***X*** *inputs or* ***W*** *variables of a module, find the values that must be assigned to some of the unspecified variables in* ***X*** *such that a* D *or a* $\bar{D}$ *will appear at one of the unspecified output variables of the module (i.e. at an element of* ***Y****).*†

Let v_i be the input or auxiliary variable carrying a D or $\bar{D}$. Find an unspecified output variable y which has v_i as one of its inputs. Using the process of decomposition we will determine the set of potential experiments in the fault free case, List1, and another set in the faulty case, List2, as described in the last section. Then we will intersect each experiment in List1 having an output value 1(0) with each experiment in List2 having an output value 0(1). If such an intersection exists, a D($\bar{D}$) can be propagated to the output variable y by selecting the variables as specified in the intersection. The process is repeated for all the module's unspecified output variables to generate all the possible solutions to the problem at a local level (note that some of these solutions may lead to an inconsistency elsewhere in the circuit). If all the attempts fail the process will terminate.

In view of the above discussion we will formulate a general algorithm that will generate all the possible solutions to the D-propagation problem. The main algorithm, D-PROPAGATION, invokes a subalgorithm SINGLE-DRIVE. This subalgorithm determines all the possible solutions to the problem of propagating the error signal (D or $\bar{D}$) to a particular output line of the module.

Let us now formulate the SINGLE-DRIVE subalgorithm.

Procedure SINGLE-DRIVE (T, e, y, Solutions, m). Given a module test cube **T** at least one D or $\bar{D}$ signal, and the set of experiments **e** describing the unspecified variable with label y, this procedure determines the signal values that should be assigned to the unspecified module variables in order to propagate a D or $\bar{D}$ to variable y. If more than one solution exists all of them will be generated. Each solution is generated in the form of a new module test cube and it will be stored in a list called Solutions. The number of solutions generated is denoted by m.

```
T1 ← T in the fault free case (Replace all the Ds with ones and all the D̄s with zeros);
T2 ← T in the faulty case (Replace all the Ds with zeros and all the D̄s with ones);
j ← 0; k ← 0;
Repeat for every experiment α ∈ e;
  IF T1 ∩ α ≠ φ
  THEN DO;
    j ← j + 1; List(j) ← T1 ∩ α

    END;
  IF T2 ∩ α ≠ φ
  THEN DO;
    k ← k + 1; List 2(k) ← T2 ∩ α;
    END;
  END;
m ← 0;
Repeat for i = 1, 2, . . . , j;
  Value ← List 1(i, y);
  Repeat for 1 = 1, 2, . . . , k;
    IF List 2(1, y) = Value, where (0̄ = 1, 1̄ = 0)
    THEN DO;
      β ← List 1(i) ∩ List 2(1) ignoring all the positions
        carrying D or D̄ in T, as well as the yth position
      IF β ≠ φ
```

†The D-propagation problem is defined differently in [9], as the problem of finding a sequence of input vectors—rather than one input vector—that when applied to the module propagates an error signal(s) from its inputs to one of its outputs. We recall that, in our definition of a functional module, we assumed that the state of every module's memory element is one of the module's outputs. Thus, it is never necessary to apply more than one input vector to a functional module in order to propagate a D or a $\bar{D}$ through it.

```
        THEN DO;
          M ← m+1;
          Solutions (m) ← β ∩ T;
          IF Value = 1
          THEN Solutions (m, y) ← D;
          ELSE Solutions (m, y) ← D̄;
          END;
        END;
      END;
    END;
  RETURN;
  END;
```

The above procedure uses five local variables, T1, T2, List1, List2, and Value. T1 and T2 represent the initial module test cubes in the fault-free and the faulty cases respectively. The two matrices List1 and List2 contain the potential experiments of the module in the fault-free and the faulty cases, respectively. By ignorning all the positions that have the value D or $\bar{D}$ initially in **T**, if we can intersect an experiment from List1 that assign Value to variable y with an experiment from List2 that assign the complement of Value to variable y, then the resulting intersection represents a solution to the D-propagation problem.

Now we will present the main D-PROPAGATION algorithm.

Procedure D-PROPAGATION (*T, E, Complete-Solutions,* λ). Given a module test cube **T** that contains at least one D or $\bar{D}$, and the complete set of experiments describing the module **E**, this procedure generates the matrix Complete-Solutions which contains all the possible solutions to the problem of propagating the error signal to one of the module's outputs. λ denotes the number of solutions generated by the procedure.

```
λ ← 0;
  Push T into Stack(1);
  Repeat for i = 1, 2, ..., n_w;
    Repeat while Stack(i) is not empty;
      Pop Stack(i) into C;
      Push C into Stack(i+1);
      IF C(n_x + i) = x and one of its inputs carries a D or D̄
      THEN DO;
        Call SINGLE-DRIVE (C, e_i, n_x + i, Solution, m);
        Push Solutions into Stack(i+1);
        IF w_i is an output variable
        THEN DO;
          place Solutions into Complete-Solutions;
          increment λ by m;
          END;
        END;
    END;
  END;
RETURN;
END.
```

The above procedure uses a set of stacks. Stack(i) is used to store all the test cubes that are candidates for propagating the error signal to w_i. Any test cube in the list Complete-Solutions can be regarded as the new module test cube after propagating the error signal to one of the module outputs. If λ is found to be 0 at the end of the algorithm, this means that it is not possible to propagate the error signal through the module.

Note that we can simply modify the above procedure to generate only a specific number of solutions, or to generate the solutions that propagate the error signal to a specific output variable. The following example will illustrate the behavior of the D-propagation algorithm.

Example 6. Consider the 4-bit up/down counter described in Examples 2 and 4. Assume that the counter variables are partially specified as follows:

$$(\mathrm{Cl},\ \mathrm{U},\ \mathrm{G},\ \mathrm{L}) = (0, \mathrm{x}, \mathrm{D}, \mathrm{x}),\ (A_1, A_2, A_3, A_4) = (1, \mathrm{x}, \mathrm{x}, \mathrm{x}),\ (Q_1, Q_2, Q_3, Q_4) = (0, \bar{\mathrm{D}}, 1, 1),$$

We recall that implication should be performed before the D-propagation operation. This results

in implying the values of c_2, b_2, c_3 and b_3 to be 0, D, 0 and 0, respectively. Hence, the initial module test cube is defined as follows:

$$\mathbf{T} = (0\,x\,D\,x\,1\,x\,x\,x\,0\,\bar{D}\,1\,1\,x\,x\,0\,D\,x\,0\,0\,x)$$

The following set of activities will occur while executing the D-PROPAGATION algorithm.

(1) Try to propagate an error signal to Q_1^*. List1 contains experiments number 2 and 3 of e_1, while List2 contains experiments number 2 and 5 of the same set of experiments. Intersecting any experiment that have output 1(0) from List1 with all the experiments that have ouput 0(1) in List2 is not possible. Hence, we cannot propagate an error signal to Q_1^*.

(2) Try to propagate an error signal to Q_2^*. List1 contains experiments number 1, 2 and 3 of e_2 while List2 contains experiments number 1, 2 and 6. There is one possible solution found by intersecting experiment 3 with experiment 6 as follows:

$$\mathbf{T} = (0\,x\,D\,1\,1\,x\,x\,x\,0\,\bar{D}\,1\,1\,x\,\bar{D}\,0\,D\,x\,0\,0\,x)$$

(3) Try to propagate an error signal to Q_3^*. In this case List1 contains experiments number 1, 2 and 4 of the set e_6 that describes Q_3^*, while List2 contains experiments number 1, 2, and 6 of the same set. No solution could be found in this case to propagate the error signal to Q_3^*. Similarly, no error signal can be propagated to Q_4^* and the whole process terminates as no more solutions can be found. Only one solution has been found that propagates a $\bar{D}$ to Q_2^* as described in step 2 above.

Line justification

The line justification problem is defined as follows: *Given a module* **M** *that has some output variables specified to be* 0 *or* 1 *and whose input variables are not completely specified, assign appropriate values to the unspecified inputs so as to justify the values of the specified output variables.*†

As in backward implication, we will justify the module output variables starting with the one which has the highest label and we will proceed backward until we justify all the module output variables. There must be more than one solution to the line justification problem (we assume that implication has been performed on the module before attempting to justify its output variables). Our strategy is to generate all the possible solutions, however the procedure to be presented can be modified easily to place an upper bound on the number of generated solutions.

Let w_i be a module variable which requires justification and let e_i be the set of experiments describing w_i. Also let **T** be the module test cube at the current stage of time. Since we will justify the module variables in a backward direction, we can assume—without loss of generality—that all the module variables with labels higher than the label of w_i, $n_x + i$, are already justified. Now to justify the value of w_i we will intersect **T** with each experiment in the set e_i. The resulting set of cubes, Solutions, contains all the possible solutions to the subproblem of justifying w_i.

It should be noted that the process described above is only valid when **T**, or more precisely that part of **T** which affects w_i does not contain a D or $\bar{D}$. If this is not true we have to justify w_i using decomposition.

Since we want to generate all the possible solutions to the original problem, we will select one of the cubes in Solutions as the new module test cube and try to justify the next unjustified variable. The other cubes in Solutions—if any—are placed on a stack, Cubes-Stack as they will be used later to generate other solutions to the main problem.

The process continues by justifying the other variables of the module using the new version of **T**. As we proceed Cubes-Stack will be updated by pushing more cubes into it. When all the variables in **T** have been justified, **T** will be placed on a list, Complete-Solutions, as it represents one of the possible solutions to the main problem.

Next, we will use the cube at the top of Cubes-Stack as the new module test cube **T** that requires justification. Note that we don't have to justify all the specified output and auxiliary variables of **T** at this stage, because we have already justified some of them before pushing the cube into the stack. For example, assume that while justifying the variable w_i we found more than one possible solution, and hence, we stored these solutions—but one—in the stack. Now, when one of these

†Our definition is different from the one used in [9] for reasons similar to the ones mentioned in the last section.

solutions is being retrieved we don't have to justify either w_i or any other variable with a higher label, i.e. we have to consider only the output and auxiliary variables having labels less than the label of w_i. For this reason, we have to push into the stack together with each cube the label of the last variable justified in that cube. Thus, each record in Cubes-Stack will consist of a module test cube and an integer label.

Also note that while we are justifying one of the variables in **T**, there is a possibility of encountering a local inconsistency. If this occurs, discard the current **T** and use the cube on top of the stack as the new module test cube. If Cubes-Stack is found to be empty, the process terminates.

In view of the above discussion, we will formulate a general algorithm that will perform the line justification operation. Again, because the solution to this problem is somewhat lengthy, we will separate the tasks involved into subalgorithms. The main algorithm, MODULE-JUSTIFICATION, invokes a subalgorithm SINGLE-LINE-JUSTIFY, which generates all the possible solutions to the general problem of justifying a single module variable. SINGLE-LINE-JUSTIFY uses another subalgorithm, ONE-CASE-JUSTIFY, which generates all the possible solutions to the problem of justifying a module variable for the special case when the module test cube is free of error signals.

Let us now formulate the ONE-CASE-JUSTIFY algorithm.

Procedure ONE-CASE-JUSTIFY (T, e, Solutions, m). Given a set of experiments **e** describing a module output or an auxiliary variable whose label is **l**, and the module test cube **T** which is free of error signals in its **l** − 1 first entries, this procedure generates all the m possible solutions to the problem of justifying the variable with label **l**. These solutions will be placed on a list Solutions.

```
m ← 0;
Repeat for every experiment α in e;
  IF T∩α ≠ φ
  THEN DO;
    m ← m + 1;
    Solutions (m) ← T∩α;
    END;
  END;
RETURN;
END;
```

Next, we will formulate the SINGLE-LINE-JUSTIFY subalgorithm.

Procedure SINGLE-LINE-JUSTIFY (T, l, e, Solutions, m). Given the module test cube **T** which may contain D or $\bar{D}$, and the set of experiments **E** that describes the module variable with label **l**, this procedure generates all the m possible solutions to the problem of justifying the variable with label **l**.

```
D-Flag ← 0;
IF T contains a D or a D̄ on its first l − 1 entries
THEN DO;
  D-Flag ← 1;
  T1 ← Fault-free value of T;
  T2 ← Faulty value of T;
  END;
IF D-Flag = 0
THEN DO;
  Call ONE-CASE-JUSTIFY (T, E, Solutions, m);
  RETURN;
  END;
ELSE DO;
  Call ONE-CASE-JUSTIFY (T1, E, Solutions 1, m1);
  Call ONE-CASE-JUSTIFY (T2, E, Solutions 2, m2);
  n ← 0;
  Repeat for i = 1, 2, . . . , m1;
    Repeat for j = 1, 2, . . . , m2;
      α ← Solutions 1(i) ∩ Solutions 2(j) ignoring the
        positions that have D or D̄ in T initially;
```

```
      IF α ≠ φ
      THEN DO;
        m ← m+1;
        Solution (m) ← α;
        END;
      END;
    END;
  END;
RETURN;
END;
```

The above procedure is straightforward.

Now let us formulate the main algorithm, MODULE-JUSTIFICATION.

Procedure MODULE-JUSTIFICATION (T, E, Complete-Solutions, λ). Given a set of experiments **E** describing a module, and the module test cube **T**, this procedure generates all the λ possible solutions to the problem of line justification.

```
λ ← 0;
Push (T, n_x + n_w + 1) into Cubes-Stack;
Repeat while Cubes-Stack is not empty;
  (T, 1) ← the top of Cubes-Stack;
  Flag ← 0;
  Repeat for i = l − 1, l − 2, ..., n_x + 1 while (Flag = 0);
    IF T(i) = 0 OR 1
    THEN DO;
      Call SINGLE-LINE-JUSTIFY (T, i, e_{i−n_x}, Solutions, m);
      IF m = 0
      THEN Flag ← 1;
      ELSE DO;
        T ← Solutions (1);
        Repeat for j = m, m − 1, ..., 2;
          Push (Solutions (j), i) into Cubes-Stack;
          END;
        END;
      END;
    END;
  IF Flag = 0
  THEN DO;
    λ ← λ + 1;
    Complete-Solutions (λ) ← T;
    END;
  END;
RETURN;
END;
```

There are two main local variables used in the above procedure, Cubes-Stack, and Flag. The function of the stack, Cubes-Stack is to hold the potential cubes generated by the SINGLE-LINE-JUSTIFY procedure in order to generate all the possible solutions to the main problem. Each record in the stack consists of a module cube (vector of size $n_x + n_w$) and an integer number representing the label of the last variable justified in the cube. The variable Flag indicates whether an inconsistency has been found or not, while justifying one of the module variables. If an inconsistency has been found (Flag = 1), then the current module test cube **T** is discarded and the cube in the top of the stack is used as the new module test cube that requires justification.

6. SUMMARY AND CONCLUSIONS

In this paper we have laid the foundation for a new test generation approach for testing LSI/VLSI chips that would treat the functional description of the chip as a parameter. Our description model bypasses the gate and flip-flop level and directly describes blocks of logic (modules) according to their functions. Binary decision diagrams were used to describe the functions of the individual modules. The experiments derived from the diagrams are amenable to extensive logical analysis, and they are especially suited for testing purposes. A functional level fault

model describing faulty behavior in the different modules of an LSI/VLSI chip was also presented. This model is quite independent of the details of implementation. The model covers functional faults that alter the behavior of a module during one of its modes of operation. It also covers stuck-at faults affecting any input pin, output pin, or interconnection line inside the chip.

A functional test generation procedure based on path sensitization was presented that takes the module level model of the chip and the functional description of its modules as parameters and generates tests to detect faults in the fault model. As in the D-algorithm, our procedure employs three basic operations namely implication, D-propagation, and line justification. These operations have to be performed on functional modules. The algorithms which perform the three basic operations on the functional modules were also presented.

We believe our approach provides a viable and effective way towards generating test sequences for LSI/VLSI circuits [1]. Moreover, our test generation technique is directly applicable to boards of MSI, and SSI chips, where each chip will be modeled as one functional module. Generating tests for circuit boards containing LSI/VLSI chips can be handled by modeling each such chip as a number of interconnected modules.

REFERENCES

1. M. S. Abadir and H. K. Reghbati, Functional test generation for digital circuits described using binary decision diagrams. *IEEE Trans. Comput.* **C35,** 375–379 (1986).
2. H. K. Reghbati, *VLSI Testing and Validation Techniques*, IEEE Computer Society Press, Washington, D.C. (1985).
3. M. S. Abadir and H. K. Reghbati, Functional specification and testing of logic circuits. *Comput. Math. Applic.* **11**(12), 1143–1153 (1985).
4. S. B. Akers, Binary decision diagram. *IEEE Trans. Comput.* **C27,** 509–516 (1978).
5. S. B. Akers, Functional testing with binary decision diagram. *Proc. 8th Int. Symp. Fault Tolerant Computing*, pp. 82–92 (1978).
6. S. B. Akers, Test generation techniques. *Computer* **13,** 9–15 (1980).
7. R. P. Batni and C. R. Kime, Module level testing approach to combinational circuits. *IEEE Trans. Comput.* **C26,** 594–604 (1976).
8. M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*. Computer Science Press, Washington, D.C. (1976).
9. M. A. Breuer and A. D. Friedman, Functional level primitives in test generation. *IEEE Trans. Comput.* **C29,** 223–235 (1980).
10. J. P. Hayes and E. J. McCluskey, Testing considerations in microprocessor-based design. *Computer* **13,** 17–26 (1980).
11. B. M. Huey and F. J. Hill, Test generation using a design language. *Proc. Symp. Computer Hardware Description Languages*, pp. 91–95 (1975).
12. Y. H. Levendel and P. R. Menon, Test generation algorithms for computer hardware description languages. *IEEE Trans. Comput.* **C31,** 577–588 (1982).
13. E. I. Muehldorf and A. D. Savkar, LSI logic testing—An overview. *IEEE Trans. Comput.* **C30,** 1–17 (1981).
14. C. Robach and G. Saucier, Microprocessor functional testing. *Digest of Papers*, 1980 *Test Conference*, pp. 433–443 (1980).
15. J. P. Roth, W. G. Bouricius and P. R. Schneider, Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits. *IEEE Trans. electron. Comput.* **EC16,** 567–580 (1967).
16. T. Sridhar and J. P. Hayes, Testing bit-sliced microprocessors. *Digest of the 9th Fault Tolerant Computing Symposium*, pp. 211–218 (1979).
17. S. H. Su and Y. Hsieh, Testing functional faults in digital systems described by register transfer language. *Digest of Papers*, 1981 *IEEE Test Conference*, pp. 447–457 (1981).
18. S. M. Thatte, Test generation for microprocessors. Ph.D. Thesis, University of Illinois at Urbana-Champaign (1979).